

VLSI CIRCUITS

Prof. S. Ramachandran

Dept of Electrical Engineering

Indian Institute of Technology, Madras

Lecture – 37

Design of Memories - ROM

(Refer Slide Time: 01:51)

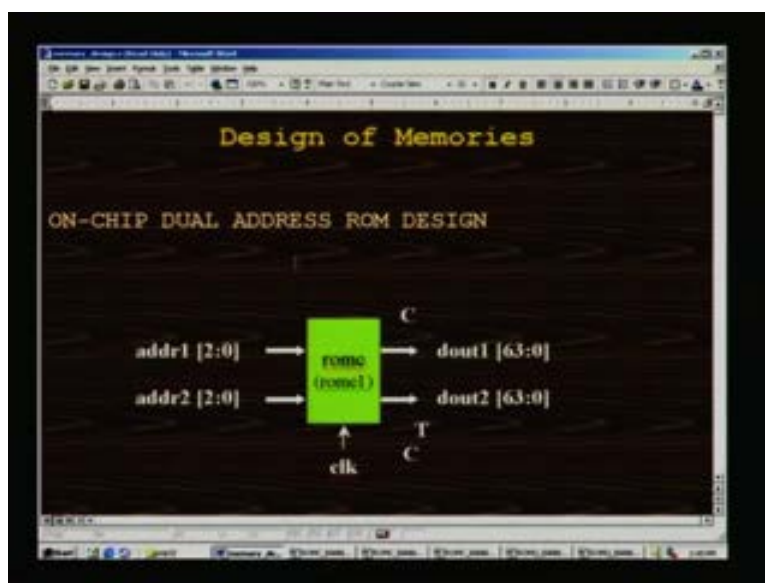


(Refer Slide Time: 02:23)



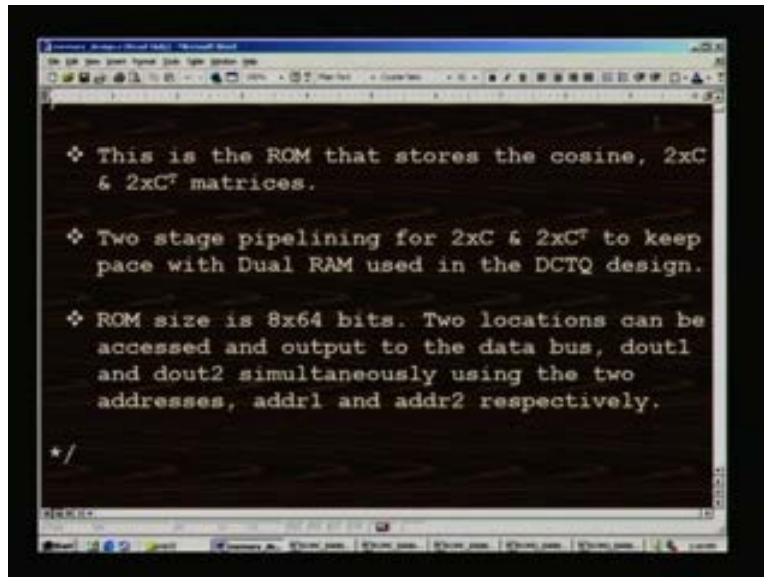
Design of memories is one of the challenging areas. We need to take extreme care while designing systems for ASIC as well as FPGA implementations. For the memories are concerned there are basically two types ROM and RAM and the organization of these memories would depend upon the typical applications.

(Refer Slide Time: 03:06)



Often the conventional memories that we use having either single address single output pattern or dual port RAM etc. may not be of help for typical applications. We are going to address that here and naturally this calls for tailor made designs for these applications. One such application that you see is right here with two addresses. For example we call this is a ROM design. It has dual address and dual data and what is implied here is and it is also synchronous that means to say it is also pipe line and we need naturally a clock as the one of the inputs and two addresses are there. We may fetch two locations it can be very same location or different locations but the number of locations that you have is actually decided by the number of bits in the address. In this case it is three bits here therefore this ROM contains eight locations totally and each of these locations is 64 bits in width and corresponding data that you need to read from the ROM table is data out one if the address one is applied at the inputs. Likewise for address 2, you need to get data out here which is also 64 bits width. Note that the ROM Content is only single. What all you have here in the side this 8 into 64 bit organization of ROM and contents are just one block as such but the two addresses are involved here. In other words, we need to access two different locations simultaneously that is the implication here and this arises in one of the design applications called discrete cosine transform and quantization. This is for used in jpeg mpeg 1 mpeg 2 video compression. We will see this design application later and for that we need this design and likewise other designs that we are going to consider such as single address ROM or dual RAM etc. which we are going to consider, will also be used in the same application. Let us see how the design flows. Here in DCT q we have what is known as a cosine term. You need a ROM for storing such cosine values and that is why the C is put here. We also need a transpose of the this constants which are put into the ROM there are basically cosine terms and we need to get a transpose of this it is basically a matrix and we need to get a transpose. This can be got by changing the address from address 1 to address 2. You can if you want C you just use only address 1 and use address 2 if you want its transpose and that is the reason why we have dual address design. Note that this is not a conventional approach say any other design would have only single address and single output either 64 bits or 8 bits and so on. Let us go into the design of this.

(Refer Slide Time: 06:45)



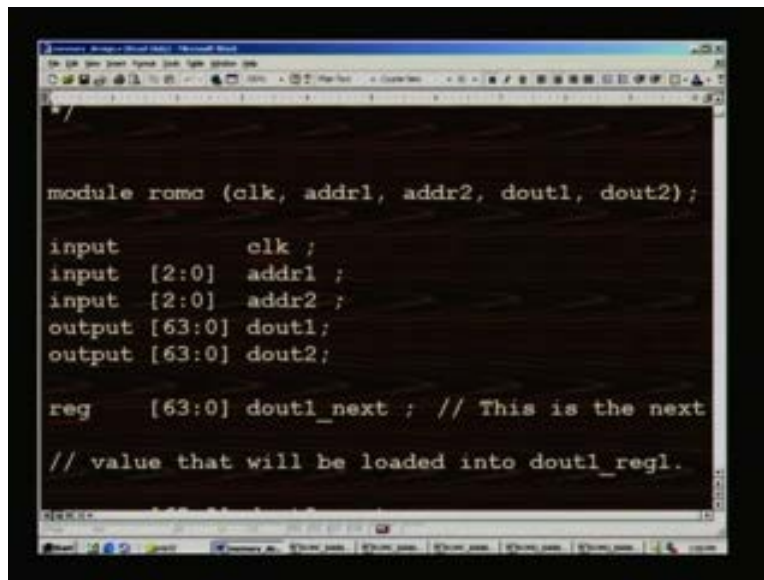
Here this is a group comment is put in this fashion and these are all bulletin here. I will just readout for your understanding. This is the ROM that stores the cosine and C and C T matrices. In fact what we need is only C and C T but for manipulation of scaling the final pixel value etc. in the DCTQ, we need multiplication factor of 2 and that being one time affair. Why not we integrate that constant? I mean multiplication also in this cosine value and the result is we put a ROM with twice the contents that we really required. This is what we really need while processing further in DCT quantization.

Next point is two-stage pipelining for the C and C T to keep pace with dual RAM used in the DCTQ design. In this case we have another specialized memory called dual RAM. We will go into the design of this as well later on for use in the same application here and mentioned here as DCTQ. In this dual RAM we need to have two-stage pipelining that means, to say the output is delayed by one clock. Whereas, in this ROM it would be enough if you just get the output with single pipelining in order to keep pace with the dual RAM we need to delay this output of ROM and that is the reason why we are having two-stage pipelining.

The third point is ROM size is 8 into 64 bits as I already mentioned. Two locations can be accessed and output to the data bus dout1 and dout2 simultaneously these are all the data bus

here and using the 2 addresses address 1 and address 2 respectively. The comment ends here and actual code starts here

(Refer Slide Time: 08:45)



```
module romc (clk, addr1, addr2, dout1, dout2);  
  
    input        clk ;  
    input [2:0]  addr1 ;  
    input [2:0]  addr2 ;  
    output [63:0] dout1 ;  
    output [63:0] dout2 ;  
  
    reg [63:0] dout1_next ; // This is the next  
    // value that will be loaded into dout1_reg1.
```

The verilog code for the ROM which is to retrieve C and C T matrices is shown here and we need to declare that design and we will name it as ROMC. C T is also implied if we want you can put C T here that does not really matter. We need to say module and finally towards the end there must be an end module as well. We have already seen that there are three inputs. One is a clock input other two are address inputs as you have already seen. Corresponding output that you want to retrieve from the ROM is dout1 and dout2 both naturally are outputs and they are 64 bits in width that is what you shown here 63 through 0 and as usual msb is written first and lsb last and separated by a colon. Similarly, the input also has its width here in this case there only 8 into 64 bits. This eights corresponds to 3 bits here as the address. During the next few lines we will see what this dout1 next is for the time being I will just mention that this a combinational signal and therefore but it is coming always within a block and therefore, we need to declare it as REG and not as wire and only in assign statements you need to use wire there. Once again this is same as the data out 1 these are all intermediate results stored and it is not really stored in a way it is stored because we use a REG here and in always block. This is the next value that will be loaded into dout1 REG1.

As I mentioned there are two pipeline stages. The first pipeline stage then the clock strikes that is at the positive edge of the clock this dout Reg1 will get the content of dout1 next. The next clock pulse that is the line second pipeline stage you will get the final output itself which is the desired output: that is the dout1 and dout2 based upon what address is contained in address 1 and address 2 which we have already seen here. This we have already covered. They are all basically registers here and this one is although it is a combinational signal, because it is put into the always block we have to declare it as REG that is we have seen before.

(Refer Slide Time: 11:26)



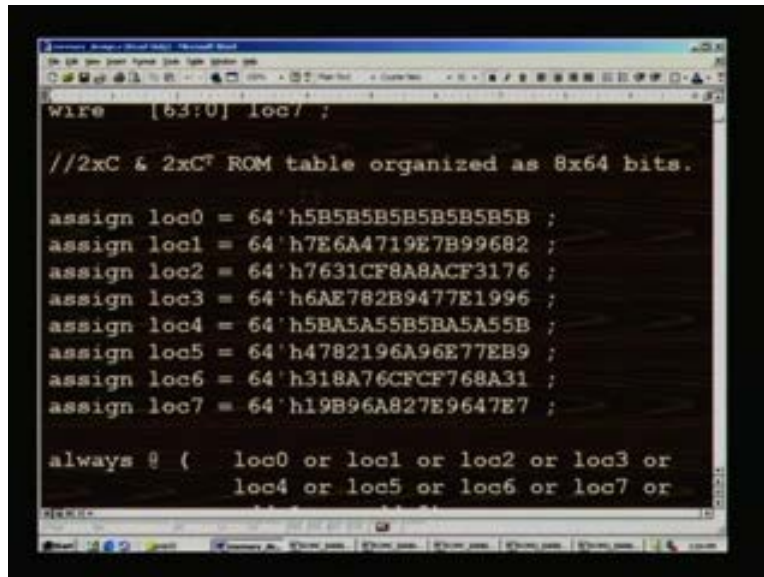
```
reg    [63:0] dout2_reg1 ;
reg    [63:0] dout1;
reg    [63:0] dout2;

wire   [63:0] loc0 ;
wire   [63:0] loc1 ;
wire   [63:0] loc2 ;
wire   [63:0] loc3 ;
wire   [63:0] loc4 ;
wire   [63:0] loc5 ;
wire   [63:0] loc6 ;
wire   [63:0] loc7 ;

//2xC & 2xCT ROM table organized as 8x64 bits.
```

In addition to this we are also going to use corresponding to eight locations we have seen that the memory contains eight different locations each of which is 64 bits. We need to declare those locations as well. They are named as loc0 through loc7 and each bit size width is mentioned here 63 colon 0 64 bits and they are all using assign statements. We need to declare them as wire here.

(Refer Slide Time: 12:05)

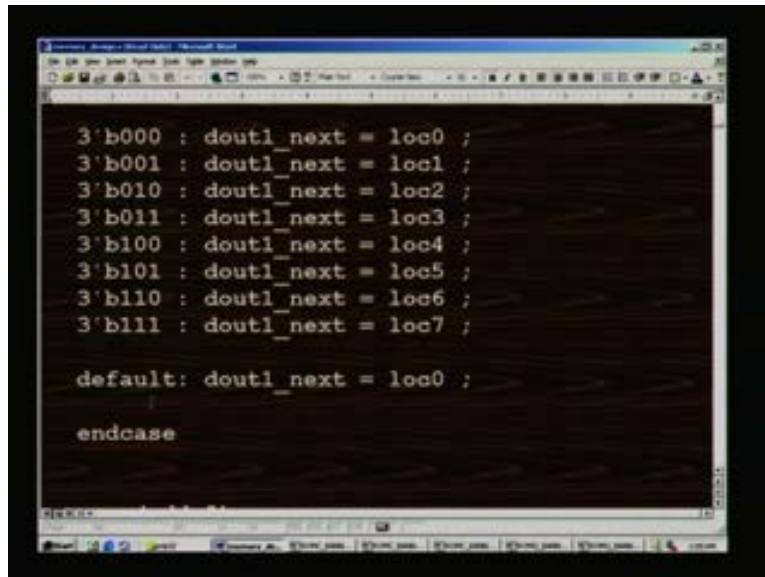
A screenshot of a Verilog code editor window. The code defines a 64-bit wire named 'loc' and initializes it with a ROM table. The ROM table consists of eight 64-bit entries, each assigned to a location variable (loc0 through loc7). The entries are hexadecimal values: loc0 is 64'h5B5B5B5B5B5B5B5B, loc1 is 64'h7E6A4719E7B99682, loc2 is 64'h7631CF8A8ACF3176, loc3 is 64'h6AE782B9477E1996, loc4 is 64'h5BA5A55B5BA5A55B, loc5 is 64'h4782196A96E77EB9, loc6 is 64'h318A76CFCF768A31, and loc7 is 64'h19B96A827E9647E7. A comment indicates that the ROM table is organized as 8x64 bits. The code also includes an 'always' block that lists the location variables.

```
wire [63:0] loc ;  
  
//2xC & 2xCT ROM table organized as 8x64 bits.  
  
assign loc0 = 64'h5B5B5B5B5B5B5B5B ;  
assign loc1 = 64'h7E6A4719E7B99682 ;  
assign loc2 = 64'h7631CF8A8ACF3176 ;  
assign loc3 = 64'h6AE782B9477E1996 ;  
assign loc4 = 64'h5BA5A55B5BA5A55B ;  
assign loc5 = 64'h4782196A96E77EB9 ;  
assign loc6 = 64'h318A76CFCF768A31 ;  
assign loc7 = 64'h19B96A827E9647E7 ;  
  
always @ ( loc0 or loc1 or loc2 or loc3 or  
loc4 or loc5 or loc6 or loc7 or
```

We have mentioned that this ROM is that... what we are implanting is only a ROM and which contains two times the cosine matrix as well as or two times the transpose of the matrix. Both imply the same contents.

The content of the ROM is precisely what you see here and the very first location is here and the msb is this one the 5B here and total bit is 64 bit. Therefore, it is declared in this fashion it is straight away represented in hex decimal. We had used assign statements here that is the reason why we declared all this yellow c has wire here.

(Refer Slide Time: 13:04)



```
3'b000 : dout1_next = loc0 ;
3'b001 : dout1_next = loc1 ;
3'b010 : dout1_next = loc2 ;
3'b011 : dout1_next = loc3 ;
3'b100 : dout1_next = loc4 ;
3'b101 : dout1_next = loc5 ;
3'b110 : dout1_next = loc6 ;
3'b111 : dout1_next = loc7 ;

default: dout1_next = loc0 ;

endcase
```

We need to this is only a combinational logic portion here and whenever this location changes only then we need to make the assignment and while doing the assignment we will use the case. We want to readout from one of the locations. Each of the location is 64 bit which is a ROM content and depending upon the address that we supply whether the address is addr 1 or addr 2 which is followed this. Depending upon the actual address value which is 3 bits here and this is binary and 0 0 0 corresponds to the location 0.

That is being read and put into dout1 next and so is the case for all other locations it all depends upon what address we encounter. Note that only one address can come at a time. It will be one of this that will be processed not all of them and that was for address one and there is also a default in order to cover do not cares or tri state and once again it's a dummy statement. At random I have allocated this. This is not going to come in the regular processing anyway yet we have to take care and in order we started with a case so we had to end case here and we start one more case within the same always block and this time we will have the address too.

(Refer Slide Time: 14:24)

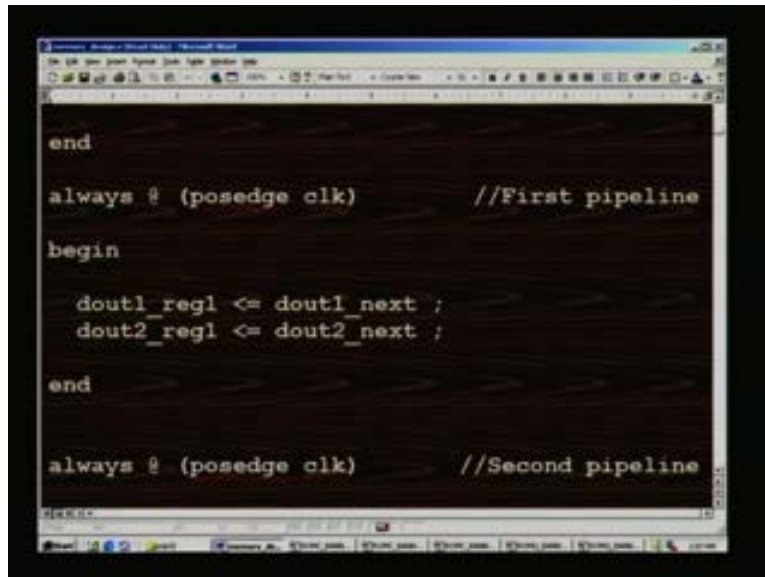


```
case (addr2)
3'b000 : dout2_next = loc0 ;
3'b001 : dout2_next = loc1 ;
3'b010 : dout2_next = loc2 ;
3'b011 : dout2_next = loc3 ;
3'b100 : dout2_next = loc4 ;
3'b101 : dout2_next = loc5 ;
3'b110 : dout2_next = loc6 ;
3'b111 : dout2_next = loc7 ;

default: dout2_next = loc0 ;
```

This is the reason why we are in a position to access two different data, namely data out 1 and data out 2 simultaneously. All this verilog statements are in general concurrent and naturally this case and that case will be concurrently that is simultaneously processed. It is exactly the same as the previous one location 0 is allocated now to dout2 instead of dout1 because this happens to be the address 2. Once again only one of these addresses will be processed at a time and whatever is the address and it will process that particular statement as such. It will retrieve that particular location and each of these locations is 64 bit in width and once again we have a default and then end case here. We started with a beginning as always so there is an end here.

(Refer Slide Time: 15:25)



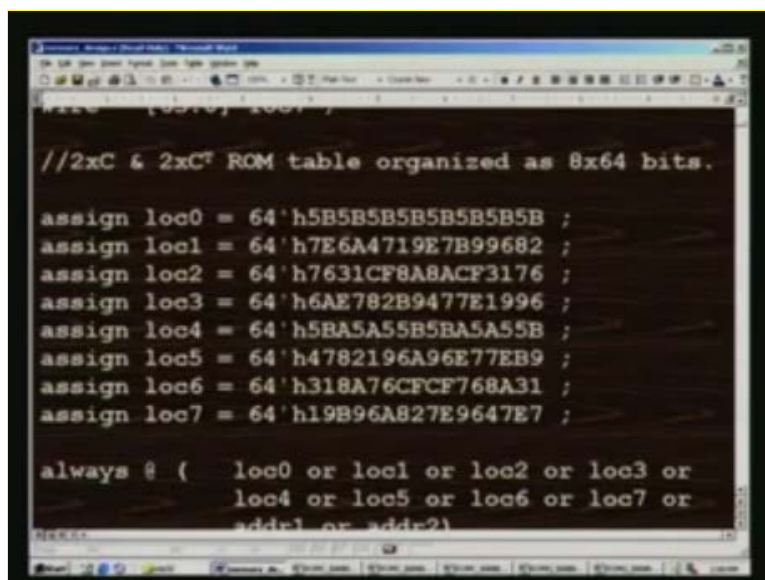
```
end

always @ (posedge clk)      //First pipeline
begin
    dout1_reg1 <= dout1_next ;
    dout2_reg1 <= dout2_next ;
end

always @ (posedge clk)      //Second pipeline
```

This is the first pipeline stage and we need to use only positive edge clock note that no reset has been used here and at positive edge of the clock, we have already seen that we have retrieve the from the ROM table this is the ROM table. (Refer Slide Time: 15:44) We had two tables what we have already seen and these are all the contents of the ROM.

(Refer Slide Time: 15:54)



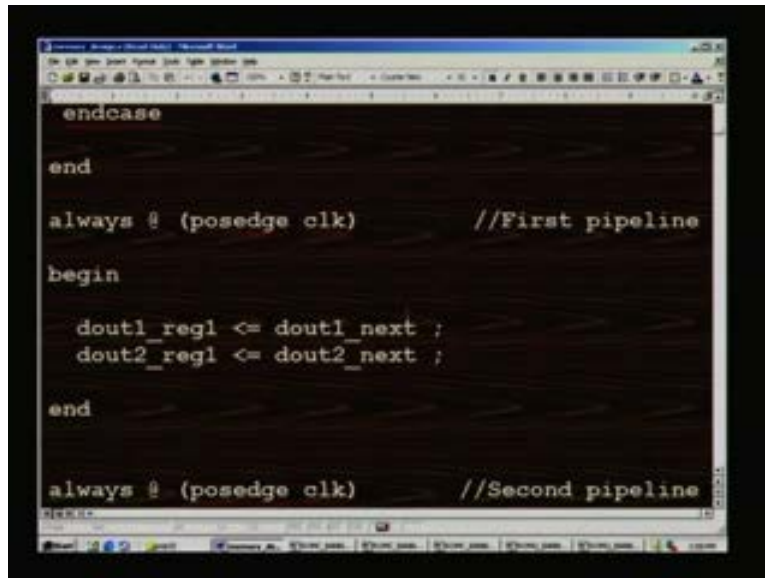
```
//2xC & 2xCT ROM table organized as 8x64 bits.

assign loc0 = 64'h5B5B5B5B5B5B5B5B ;
assign loc1 = 64'h7E6A4719E7B99682 ;
assign loc2 = 64'h7631CF8A8ACF3176 ;
assign loc3 = 64'h6AE782B9477E1996 ;
assign loc4 = 64'h5BA5A55B5BA5A55B ;
assign loc5 = 64'h4782196A96E77EB9 ;
assign loc6 = 64'h318A76CF768A31 ;
assign loc7 = 64'h19B96A827E9647E7 ;

always @ ( loc0 or loc1 or loc2 or loc3 or
           loc4 or loc5 or loc6 or loc7 or
           addr1 or addr2)
```

Whatever the address in address 1 or address 2 that will be retrieved, and that corresponding location will be retrieved. The data that is retrieved is precisely one of these and assign to this location 0 through 7 depending upon what address depending upon the address that we give. We have already mentioned that simultaneously you can mention two addresses and get two data as such. The data what we get is 64 bit which we have seen just now.

(Refer Slide Time: 16:33)



```
endcase
end

always @ (posedge clk) //First pipeline
begin
    dout1_reg1 <= dout1_next ;
    dout2_reg1 <= dout2_next ;
end

always @ (posedge clk) //Second pipeline
```

This is the first pipeline that we have already seen. We had just now seen 2 case statements. The output of which is this dout1_next and this is assigned to dout1_REG1. This is the first pipeline stage. Earlier in the case what we have had is only a combinational circuit and retrieving the ROM data and that ROM data when a positive edge of a clock is encounter it is assigned to dout REG1. This is once again intermediate output and this will take it as the input for the next clock edge.

(Refer Slide Time: 17:13)



```
always @(posedge clk) //second pipeline

begin

    dout1 <= dout1_reg1 ;
    dout2 <= dout2_reg1 ;

end

endmodule

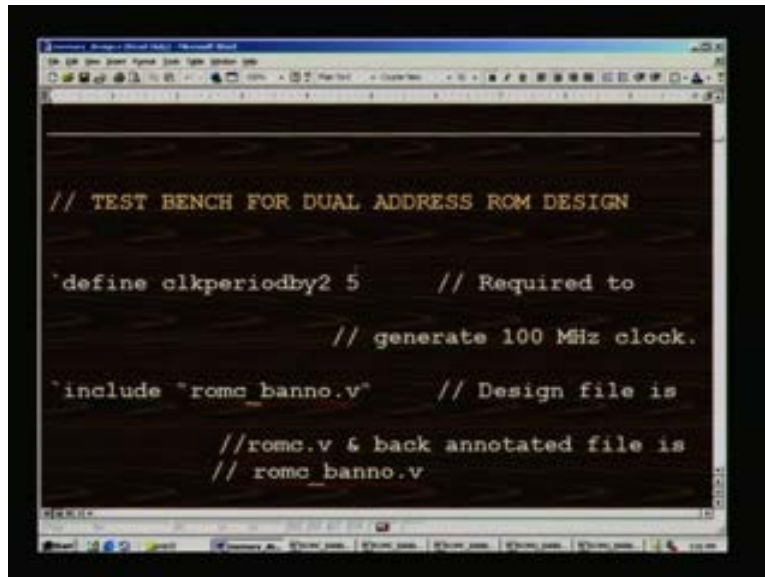
// TEST BENCH FOR DUAL ADDRESS ROM DESIGN
```

This is the second pipeline. Note that what we assigned here to REG1. This is the actual ROM data and we assigned here this two corresponds to two addresses. You want to simultaneously access two locations that is the idea here. This is assigned here at when the clock strikes and positive edge of course. This is the final output that we have so dout1 and dout2 are the two data corresponding to the final output. That is the beginning therefore there is an end there is an end module here.

This is what we had here. (Refer slide time: 17:59) This was the RAM here with two addresses. We wish to at simultaneously retrieve two locations and each of this location is 8 bytes in width. This is the requirement for DCTQ design application that is the reason why we went for an unconventional ROM implementation, which is possible on FPGA as well as ASIC because number of I mean chip area is not too much here in this case. If the chip area is increases of the order of several kilo bytes still you can accommodate in high capacity FPGAs. There is a limit for all this. Even in ASIC we will have problem because later on we will learn while designing dual RAM that here is a statement which invokes register arrays actually register is nothing but flip flops. Those are invoked so fundamentally they are all flip flops. So, there is a limit both in FPGA as well as ASIC design. In ASIC however there are specialized vendors who have ample design experience on custom built memories. Such vendor cores can be bought and integrated into your design, if your design requires either the conventional RAM or ROM or tailor made

custom built for your application. In FPGA we have this more limitation because you cannot go beyond several tones of kilo bytes of memory. This fortunately is not much. 8 into 64 bits only are being used and that is reason why we have gone for this coding.

(Refer Slide Time: 20:00)



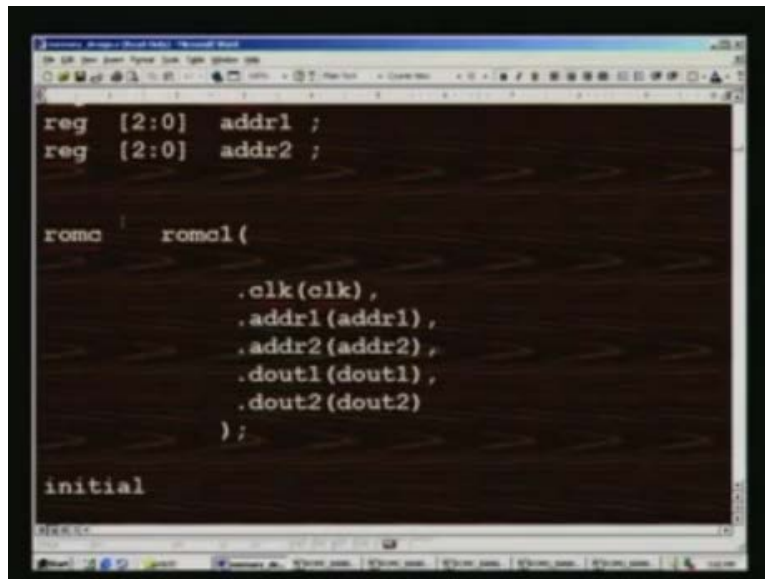
```
// TEST BENCH FOR DUAL ADDRESS ROM DESIGN

`define clkperiodby2 5      // Required to
                            // generate 100 MHz clock.

`include "romc_banno.v"    // Design file is
                            //romc.v & back annotated file is
                            // romc_banno.v
```

Let us see the test bench now. The code is quite simple here. This is the test bench for testing this for dual address RAM design which we have just now covered. Once again we have a clock operating at hundred megahertz that is why we put a 5 here. This is a back annotated file and corresponding back annotated waveforms also you will be seeing shortly. That is why we have included ROMC back annotated dot v file which is precisely the same as ROMC dot v except that this has been obtained after back annotation. The test module is ROMC **underscore** test that is why we have to declare it here and herein what all we are interested is only the data outputs and which is declared here and inputs in test bench as I mentioned will have to be declared as to be REG

(Refer Slide Time: 20:50)



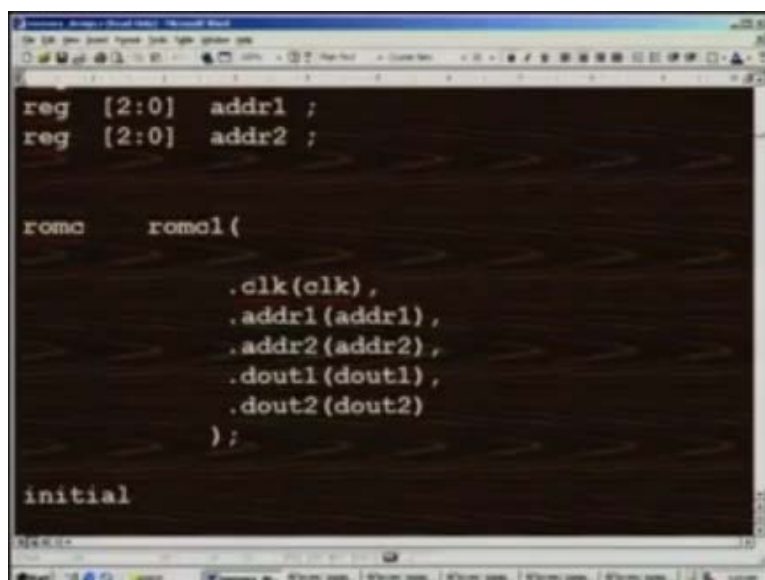
```
reg [2:0] addr1 ;
reg [2:0] addr2 ;

romc romc1 (
    .clk(clk),
    .addr1(addr1),
    .addr2(addr2),
    .dout1(dout1),
    .dout2(dout2)
);

initial
```

Now we know the design ROMC here and module is still ROMC even back annotated file. If you open it out, you will see the same module name which we have used in our design and this is the instantiation. If you want to call many more times you can just change this one and once again port name we are using. We can put it in any order and this are all the inputs here clock and they are all synchronous ROM here and two addresses are there and as inputs and corresponding outputs manifest in dout1 and dout2 respectively.

(Refer Slide Time: 21:34)



```
reg [2:0] addr1 ;
reg [2:0] addr2 ;

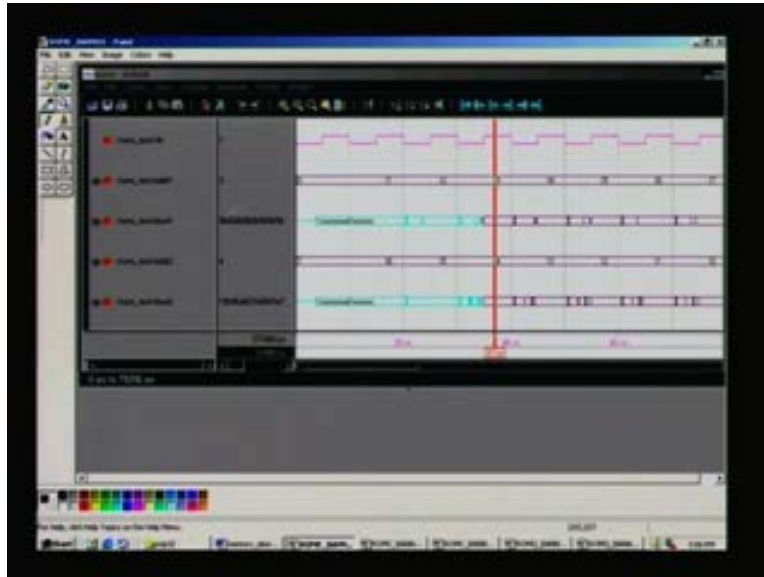
romc romc1 (
    .clk(clk),
    .addr1(addr1),
    .addr2(addr2),
    .dout1(dout1),
    .dout2(dout2)
);

initial
```

Being a test bench we will have to take appropriate action with reference to time. We can use the initial statement. There will be a beginning and corresponding end will be there towards the end and we need to initialize the clock. Here we have two addresses so what we will do is we keep changing different points of time the addresses. To start with address 1 and address 2 or initialize to different values. For example this is 0 and this is seventh location this corresponds to location 0 and this is location 7. That means to say when you process this you can access location 0 as well as location 7 simultaneously and get it at the outputs dout1 and dout2 respectively that is the implication here. This is deliberately done into I mean different addresses are given here. You will get a feel of it when we see the waveform. Here I just tag at the address bit by 7 nanosecond in order to avoid very first clock. First clock edge happens at 5 nanoseconds. 0 starts. At 0 time and so I want to avoid first clock so that remember that we have two stage of pipelining. That means to say the data can manifest only after the third clock. That is the implication there and that is the reason why we have staggered a bit. There is an off-set between the address change and within the clock edge. Clock edge occurs at 5 nanosecond and we are we want avoid this first clock so we are delaying a bit here that is the implication there. Every time you see here we applied two addresses address 1 address 2 one we will start with applying 0 then for address to 7 and keep going every 10 nanoseconds we have a hundred mega hertz operating frequency. Every 10 nanosecond we keep changing the address and corresponding data should manifest which we can observe in the waveform.

Here instead of binary we can also use a decimal here in which case you need to use d here. Straight away use 0 1 2 3 and so on. Here you notice that all are incremented as for as address 1 is concerned by 1 0 1 2 3 and so on, whereas, address 2 goes the other way 7 6 5 and so on. This happens every 10 nanoseconds. Finally when you are done you stop, give some question and then stop here. There was a begin therefore there is an end and as usual you need to toggle the clock this is the standard statement we have been using all through that is with the all the statement here. (Refer Slide Time: 24:24) This data was 5 therefore 10 nanoseconds with the time period therefore hundred megahertz. You started with a module for the test bench therefore there must be an end module.

(Refer Slide Time: 24:57)



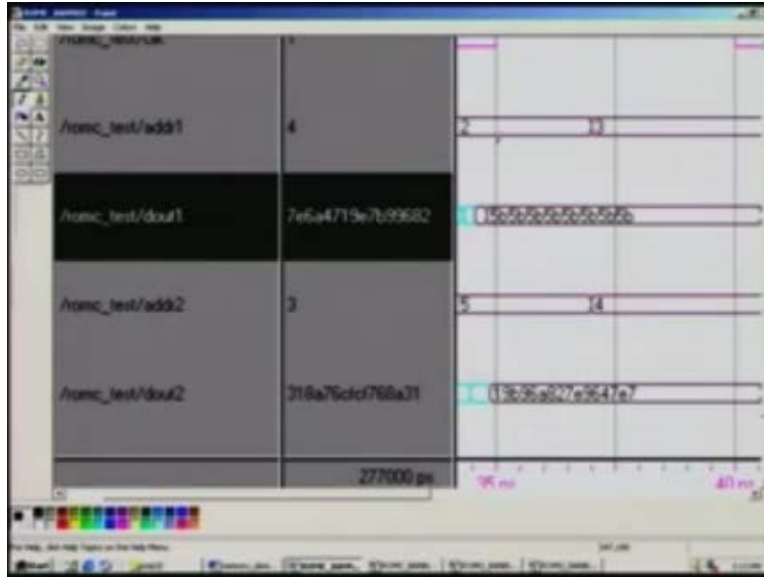
Before we go into the synopsis result, let us have a look at the waveform for this here. I will first explain this and then zoom. What we have is a clock here and you can see the waveform on the clock and address 1 is here and corresponding data out 1 is mentioned here and address 2. I have kept it deliberately different thing just to get a feel that different addresses are working. You can repeat the same xys by using the same address. I have cross checked that also that is also working fine there is no problem because the statements are very plain. Verilog statements are straight away case so there is 1 to 1 correspondence and notice this one as I mentioned there is a clock here raising at 5 nanosecond this is 20 nanosecond therefore this 10 and when you this is midway here. This is 5 nanoseconds of time so that is what the clock is 100 megahertz. It should amount to 10 nanoseconds and we apply the address right here. Actually the 7 nanoseconds only we apply and by default this simulator has I think clearing it does all by itself. That may be the reason why that is, but it is not really true because 7 is also coming here and this is little not applicable because we have given at 7 nanoseconds only if you remember. (Refer Slide Time: 26:46) The test bench - let us see no 7 nanoseconds we are writing the same information once again. Address 1 address 2 0 and 7 are already done at 0 nanoseconds. You have to take care just always verify crosscheck then you will see what the reason is. (Refer Slide time: 27:10) That explains why it is 0 and so here and actually we actually start only after 7 nanoseconds. After 17 nanoseconds only so the address will be changed that is the meaning there because as for the this

one we have changing at 7 the cumulative time you take 7 plus 10 17 nanoseconds only we apply the second address here (Refer Slide Time: 27:48) and that is what is happening here.

This address is happening here so you should not take this as the first clock that is what I am trying to impress upon you. The first clock actually is this 1 and second clock is this. Let us see where the output is dout1 corresponds to the address 1 and as I mention it is a two stage pipelining. It should manifest the result must manifest only after the end of 2 clock pulses not counting this first one. First one happens here, then second here, then third one is going to occur here but the data is stable right here. The corresponding data here is if you read here it is 5 b. Let us see the table here. Just remember 5 b here and similarly address 2 is 7 and corresponding data is what you see here is 19 b9 and so on. It is precisely at this point of time 37000 picoseconds is 37 nanosecond right at this point of time where the curve series. Just remember this at least few numbers here all 5 bs here and 19b9 some 47e7 and let us see what we have given here. (Refer slide Time: 29:13) The design if you go so this is the data that we have given. If you remember 5 b etcetera and other 1 was 19b9 in it 47e7. Location 7 is this one location 0 is this one. I think that is what we have been looking at. dout1 is corresponding to address 1 which is 0. 0 address data is received only after 2 clock pulses delay so because of the pipelining here and you are there are too crowded so many hexadecimal 8 bytes are there. You cannot obviously shown all of them therefore I just clicked on the cursor and that cursor data is being displayed here which is adequate.

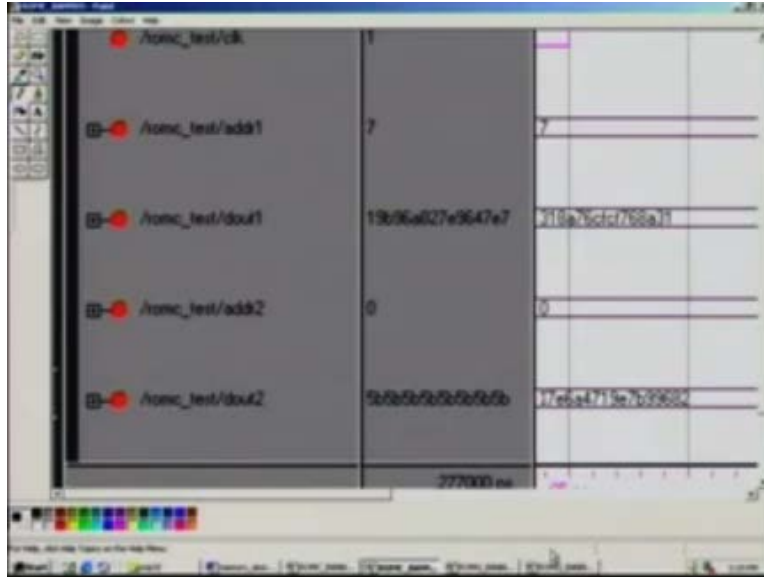
Note that already the address has gone quite ahead because of the pipelining. Remember this data is for 0 address and so is the case for address 7 the data corresponding data is 1 9 that is what we had this was the very first location there location 0 content and location 7 content corresponding to the addresses 0 and 7. This is corresponding to next few address 1 has advanced for there. I have expanded for this clock here little more and what you see here is and the cursor is at this position. It is 7 e 6 a and so on. It is precisely the same here and this is corresponding to what address. This is little tree this was 5 b. That was the first location. This location must correspond to second 1 that is address 1 here and this must this was the 7th location. This must corresponds to 6, so if you can remember this fine 7 e here then 3 1 here.

(Refer Slide Time: 31:16)



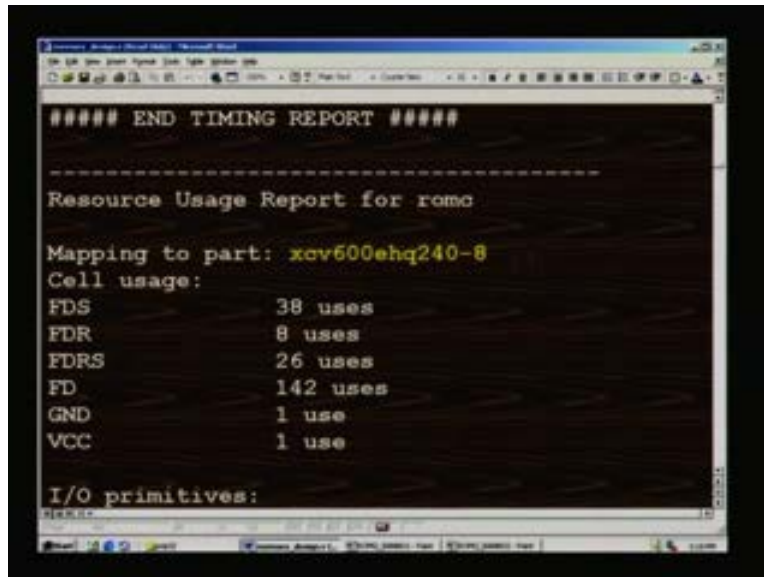
Let us see the contents here 3 1 is here 7 is here right 7 e then 3 1. Location 1 and 6 that is what you wanted. This is the address 1 and of course address we can give because of the pipelining, and this is the data that you have there. This is the first data and this is the second data. Location 1 this is corresponding to address 1 so you should have this is the data. Notice that the data is changing at too many places. Hence, this is because in our design we have put so many intermediate stages. You can probably an attempt rewriting redesigning whether to improve this. Frequency operation can be jagged up or not you can try. I am not assuring that better solution may emerge but you can try. In fact I did not try other means because the frequency was already met. That is the reason why data is changing and ultimately it will get stable. That will happen before the clock positive edge of the clock. You can see that positive edge of the clock happens here the right on the brink but still it is safe. (Refer Slide Time: 33:04) Similarly, the other data what we have seen that it is corresponding to location 6 so just remember 7e82 here 3131 that is what we had 7e corresponds to location 1. We have not mixed it up right and this is what we have seen.

(Refer Slide time: 33:40)



We will see 1 more waveform here and I will zoom this. Once again we will see look at dout1 and dout2 which is this data is 3131 which is same just overlapped. You can view and next data is here. See that (Refer Slide time: 34:08) this is the positive edge of the clock, so expanded here. The next edge of the clock at this edge this data has valid and this edge this data. We have 2 data here let us see where it corresponds to in the RAM table. One is this is 5b. All 5bs and this is 19 b9 etcetera. Address 1 is location 7 now here is that correct and this has come down to location 0, and (Refer Slide time: 35:00) because of the pipelining this addresses are advanced further advance. Do not give the addresses just take the data alone. We have seen this as well. We will just look at the simplify results and what it has to show let us see. This is the test bench we have already covered this synopsis results you see that it's a quite a high frequency of operation we have reported and here.

(Refer Slide Time: 35:49)



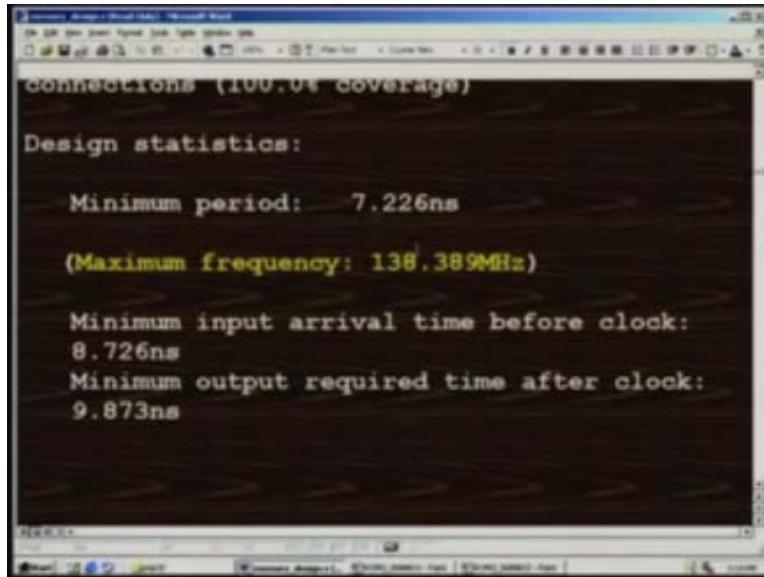
```
##### END TIMING REPORT #####
-----
Resource Usage Report for romc

Mapping to part: xcv600ehq240-8
Cell usage:
FDS          38 uses
FDR           8 uses
FDRS         26 uses
FD           142 uses
GND           1 use
VCC           1 use

I/O primitives:
```

We have mapped on to a device which we are probably going to use our DCT design on this same device. After all the designs which pertain to only the DCTQ application and such as arithmetic circuits and dual rams etcetera and we will use this same device with the possibly higher speed available. It is a 240 pin package and it's a vertex c combination and it reports different flip flops usage how many and input buffers output buffers all that. Finally it reports LUT of just 68 for this RAM application and xilinx place and route reports and these are all the slices 4 input LUTs used here and the two of interest will be around 2000 gates is a standardized two input NAND gate. Gate equivalent this particular design for a dual address. In addition to that you need additional gates if you want JTAG complaint IOC.

(Refer Slide Time: 37:04)



```
connections (100.0% coverage)

Design statistics:

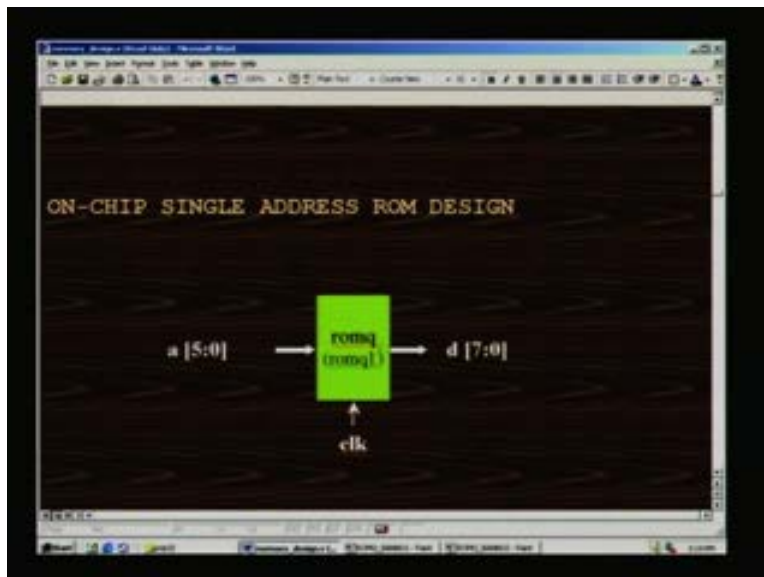
  Minimum period: 7.226ns

(Maximum frequency: 138.389MHz)

  Minimum input arrival time before clock:
  8.726ns
  Minimum output required time after clock:
  9.873ns
```

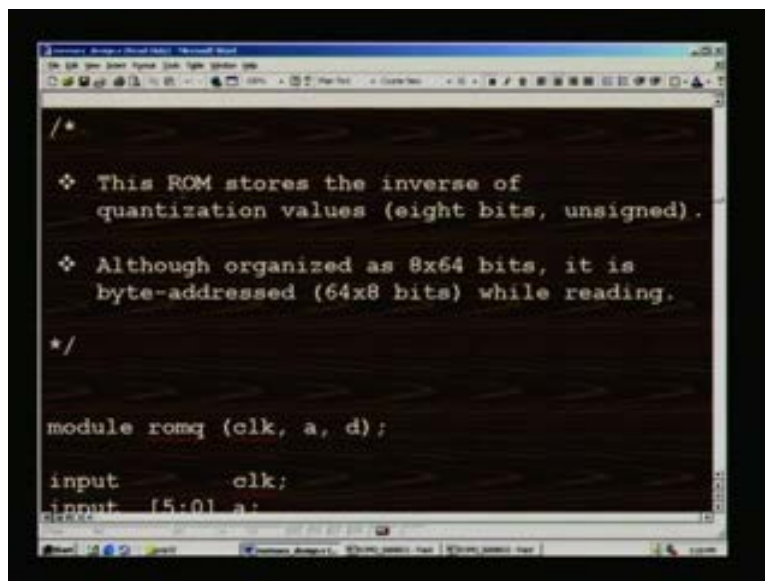
Here you see that the frequency operation has come down after place and route here so but still it is good enough but once again as I mentioned what counts is the total system speed not nearly the individual models. It is only an academic interest we are looking at this figures and it also creates a ROMC dot bit for downloading into hardware. Next we will consider 1 more design this appears to be a conventional design this also used in DCT quantization.

(Refer Slide Time: 37:38)



This is used as a quantization table so in quantization we need to carry out division not. Multiplication ultimately after getting DCT which is the transform and but division can also be implemented as multiplication if you just invert the quantization value. What is stored in the ROM table is the inverse of quantization and what is stored here is actually once again 64 bits. When you want to retrieve it we want to retrieve one byte at a time although it is 8 by 64 organizations. We want to retrieve as in a byte oriented manner and this is of from the conventional design and that is why hence we need to write a code for that and mind you it is a 64 bytes totally each is 8 bytes, eight locations so 64 bytes totally. Naturally you call for address of that size 6 bits here 2 to the power of 6 will give so many location and the design starts here and just read out the comments here.

(Refer Slide Time: 38:49)

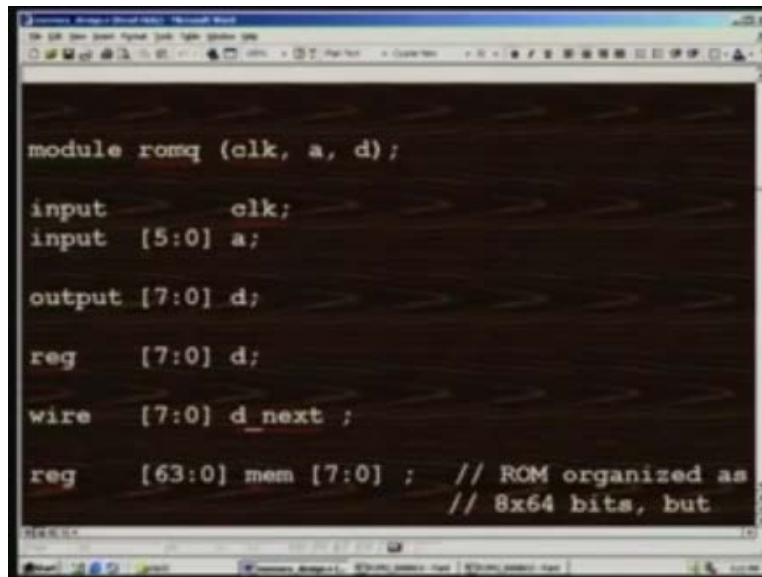


```
/*
 * This ROM stores the inverse of
 * quantization values (eight bits, unsigned).
 * Although organized as 8x64 bits, it is
 * byte-addressed (64x8 bits) while reading.
 */

module romq (clk, a, d);
input      clk;
input [5:0] a;
```

This ROM stores the inverse of quantization values 8 bits unsigned so quantization value is its an inverse of quantization value as I mentioned before. Although it is organized as 8 by 64 bits it is byte addressed while reading. While reading only that is byte address and retrieved as a byte and process for that because for we need to divide DCT value which will emerge at every clock pulse and there are 64 coefficient in that and each coefficient will have to be divided by this quantized value in order to get a quantized DCT. That is what brings about compression in which will see in greater depth later on when we talk about the design application and here the design module is quantization.

(Refer Slide Time: 39:42)



```
module romq (clk, a, d);  
  
    input      clk;  
    input  [5:0] a;  
  
    output [7:0] d;  
  
    reg  [7:0] d;  
  
    wire  [7:0] d_next ;  
  
    reg  [63:0] mem [7:0] ; // ROM organized as  
                           // 8x64 bits, but
```

Q is next here for ROM. It is called ROM q module and it has two inputs one address which we have already seen. It is a width is 6 bits and its data is 8 bits here and we declare the input outputs here and this is a register we will be using in pipeline stage once again here also and so data will be in the always positive edge block. Therefore, it is REG here and we also need intermediate assigned value which will declare as wire here and that is precisely same as d here. It is an intermediate value to store the data and now. As I mentioned memory can be realized by as flip flop array. You can just by one statement you can declare any size memory. For example this note with care here – it is retard as REG here and you give the width here. If it is 64 bit you want memory which is 8 into 64. 64 bit width that is specified first and you can declare you can give a name for the memory. For example I have given it as MEM if you want call it as something else say RAM or ROM whatever we can call any name any meaningful name you can give for your memory and these are all the locations. Say number of locations are eight so location 0 through location 7 we have tackled earlier in the same fashion and this 1 is the highest order address so locations and so on. This is the nomenclature even here first MSB then followed by the LSB here. This is typical memory realization. This is an important thing we should remember. For any memory realization you can use the array of this type and similarly one more array also has been used and ROM is organized as 8 by 64 bits and but read byte by byte. In order to assist bi-furcating this word into bytes we need one more here array here and that is what is declared here. In addition to this we need some more signal for example memory data is

also is 64 bits and they are I mean it's a combinational output. We declare it as wire here. Also we need location 0 through 7 as in the previous case each of which is of the same bits 64 bits and they are all wire and that is because we are going to use assign statement here. As before we have exactly the same only difference is data is different here so these are all the inverse quantization values showed in the RAM table and as I mentioned here it is eight numbers of locations each is 64. The organization is 8 into 64 as for as the storing (42:45) is concerned is being a RAM it is only storing device. While retrieving if you want to retrieve for local, we have 6 bit address. When we give address 0 we want to retrieve ff first. For address 1 we want to retrieve 8 0 and so on for this address 7 3 c will be retrieved then 8 0 will be retrieved at 8 address and so on. In order you can just address that. Finally for 63 address decimal 1 9 will be used. When we see the waveform we will compare come back to this. Once again this the combinational always block and whenever that locations change only then we need to do the processing. What we are doing here is we have used REG MEM array earlier. This is the way to retrieve particular memory location those who are familiar about microprocessor and the RAM design. We will follow it readily.

(Refer Slide Time: 43:51)



```
mem[1] = loc1 ;
mem[2] = loc2 ;
mem[3] = loc3 ;
mem[4] = loc4 ;
mem[5] = loc5 ;
mem[6] = loc6 ;
mem[7] = loc7 ;

end

// Bytes from each row accessed in raster
// scan order (MSB first, etc).

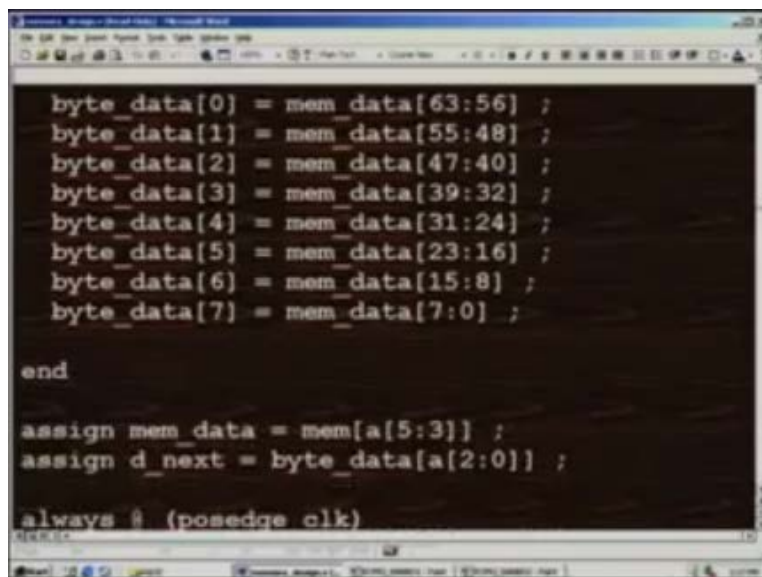
always @ (mem_data)
```

When you say memory the address you give here within the quotes. Straight away that it will go to that location take the 0 as the address and retrieve whatever is stored in that 0 location and in fact the other way. It goes to the location 0 here because it's a RAM we want to read it from the

RAM location 0 is nothing other than the data that we have already stored that is retrieved as memory 0. This is done so that address can be varied later on. We will go into the details little later so what precisely it means it will be clear when we deal with that later on.

Now the comment is here bytes from each row accessed in raster scan order MSB first. We have already seen that ff was the first one that is the access first and this is the flexibility in writing in Verilog code. Suppose you are not happy with the MSB first you want some other thing to be retrieved first. You can always write a code to change your particular custom design. That is the beauty of verilog coding here. We have one more always block and this time another variable is used memory data and notice that memory data is byte access here. For example 63 to 56 to subtract this you get 7 so plus 1 so 8 bits totally. In this order lower order bits are assigned to different byte data 0 and so on. This is only to (Refer Slide Time: 45:40) facilitate change of 64 bit into 8 bit notation.

(Refer Slide Time: 45:46)



```
byte_data[0] = mem_data[63:56] ;
byte_data[1] = mem_data[55:48] ;
byte_data[2] = mem_data[47:40] ;
byte_data[3] = mem_data[39:32] ;
byte_data[4] = mem_data[31:24] ;
byte_data[5] = mem_data[23:16] ;
byte_data[6] = mem_data[15:8] ;
byte_data[7] = mem_data[7:0] ;

end

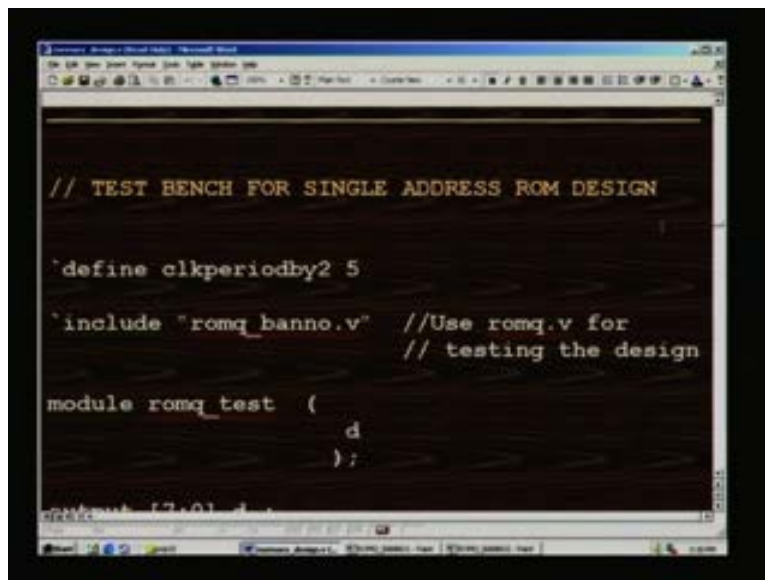
assign mem_data = mem[a[5:3]] ;
assign d_next = byte_data[a[2:0]] ;

always @(posedge clk)
```

We have used memory data earlier and we have also used memory that within brackets the address. We can straight away define the address is MSB of the address we have taken here 3 4 5 and that particular thing will fetch this memory data memory data is nothing other than what we have here. It is clearly a only a byte is being fetched here memory data here right. That is fetch from the address pointing to the MSB of that and MS LSB of the address will fetch the byte data.

This byte data then give the address this is in fact this is the byte data here and memory is what we have. When we say memory data so what it implies here let us see have a look,(Refer Slide Time: 46:49) actually we say memory data it actually implies all the 64 bits it does not mean this. This is here only in fact reassign as the byte data here separating out a 64 bit content as a byte content here. This is actually 64 bit and data byte is to get only one byte at a time and that is what it is there. That is what the two statements are doing. You retrieve the byte data and first from the one complete 8 byte location as such. What we this is single pipeline here and this. Next is nothing other than the actual retrieved byte data corresponding to a address a which is some total of all the bits 5 through 0. You have to see together the statements then only it will become clear it is addressed completely 0 through 5 addresses so, which outcomes the outcome will be 1 byte as such that is what we have here. This is the address. This byte red correspond to a address is assigned to d here and we end up this and finally the module is also end up.

(Refer Slide Time: 48:28)



```
// TEST BENCH FOR SINGLE ADDRESS ROM DESIGN

`define clkperiodby2 5

`include "romq_banno.v" //Use romq.v for
                        // testing the design

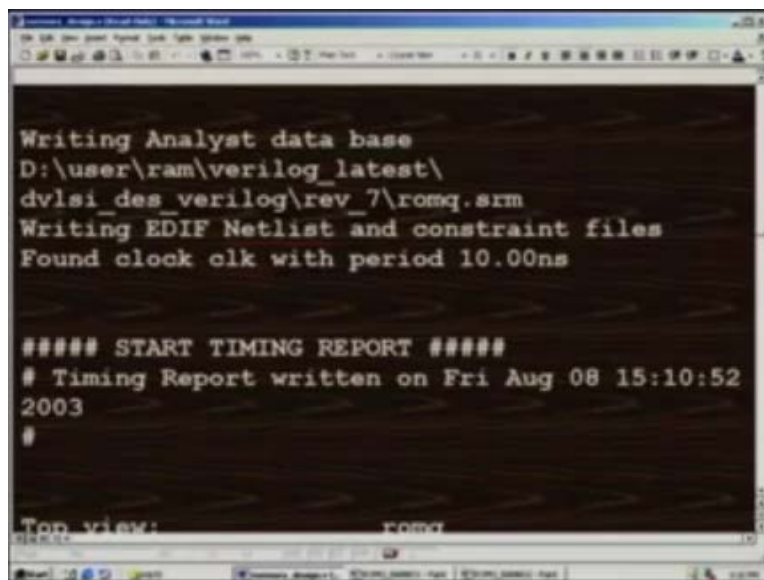
module romq_test (
                    d
                    );

output [7:0] d;
endmodule
```

This is the test bench for testing the RAM. As usual these are all the input statements and this ROMQ test here. Here what we need is only d output here and we declare as output here being only one byte and we need to have inputs declared as REG so we have it here and we instantiate the design invoke the design and instantiate and port by name and note the interesting thing here. We are using count as an integer we are declaring this is exactly a 'c' like structure. This precisely is the c program exact counter for c program. I mention that in Artial coding you

should avoid all these. We are doing using this only in the test bench. If had you use this statement for example for statement is used which is precisely a c statement and for count is equal to 0 and for countless that 64. Every time you count advance the counter. What we need to do is and assign this count to the address here that is what this for loop is doing. This exactly counter part of c and this is violation of Artial guidelines and you should not use this in the design you should you can use in the test bench. Because it gives much concise pro coding you can call it programming at for the test bench. It is a pseudo thing you can sometimes you can also use Artial coding style here although it is not really required. W can violate in the test bench but not in the design this should be very clearly keep in mind. I think I have been mentioning this right from day one, we started the verilog coding. Once having covered all these 64 addresses. We need to give little more allowance and stop here. As usual this clock is running at hundred mega hertz this statement and the end module here.

(Refer Slide Time: 50:32)



```
Writing Analyst data base
D:\user\ram\verilog_latest\
dvlsi_des_verilog_rev_7\romq.srm
Writing EDIF Netlist and constraint files
Found clock clk with period 10.00ns

##### START TIMING REPORT #####
# Timing Report written on Fri Aug 08 15:10:52
2003
#

Top view: romq
```

The results are here. It is running at 133 mega hertz here and prior to this. Let us have the waveform here. (Refer Slide Time: 50:46) Remember that there is 1 7 nanoseconds in test bench here also as in the previous example that is why address is applied from 7 nanoseconds here. There is a clock here there is this is the data out and this count is also displayed here. You notice that keeps on advancing this is to facilitate supplying address here in fact this is same thing is applied here and this is synchronous and with one pipelining.

(Refer Slide time: 52:36)



The output comes delayed after this. You can see here this is the clock which is responsible for this retrieving the data corresponding to 0. Address and the data is ff. You remember that we started with it earlier ff 8 0 that is precisely what you have here. You can just zoom. You can see ff 8 0 and so on right and notice that this is the clock which is responsible but it comes only after gate delay because we are using a back annotation here and ff 8 0 6 c and second waveform you can see here. They are all towards the end this is 62 63 2 5 1 e 1 9 etcetera. If you see the content here, you can see ff 8 0 and last 1 one e 1 9 you can see that. You can see 2 d 2 5 1 e 1 9. I can see this 2 5 2 1 8 1 9 so this establishes the working of the design and finally we will have the look at these, all the simplified results. We have seen here this works at 123 mega hertz and corresponding we have used the very same device that we have mentioned before and it has taken just 37 LUTs for this design and place and route result is 319 gates here and ROMQ dot bit is output and it operates at 152 mega hertz. Thank You.